他の言語を学んで自由になろう

山本和彦

kazu@iij.ad.jp

自己紹介

中学 Basic, Z80アセンブラ

大学 FORTRAN, Pascal

1994 Lisp, C  Mew  KAME

2006 JavaScript  Firemacs

2007 Haskell  mighttpd

プログラミング言語は
思考を支配する

「ハッカーと画家」
～ 普通のやつらの上を行け ～
Paul Graham

毎年少なくとも
一つの言語を学習する

「達人プログラマー」

Andrew Hunt and David Thomas


違う考え方を学んで
自由になろう

目的

Java プログラマに
関数プログラミング
の優れた部分を紹介する

免責

僕は Java が得意ではありません

Java のコードはぎこちないと思います

この資料を作るために
Java の本を3冊読み直したので
許して下さい

免責

例は僕が得意な Haskell で書きます

業務に活かすには Scala を
学ぶのがいいでしょう

Scala で書く余裕はありませんでした

無名関数

Java のイベントハンドラ

- 「Java 開発者のための関数プログラミング」より
  - Dean Wampler
- AWT ActionListener の定義

```
public interface ActionListener
extends EventListener {
  public void actionPerformed(ActionEvent e);
}
```

イベントハンドラを設定する

- AWT ActionListener を使ってみる

```
import java.awt.*;
import java.awt.event.*;

class ButtonApp {
  private final Button button = new Button();

  public ButtonApp() {
    button.addActionListener(
     new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        System.out.println("event received: " + e);
      }
    });}}
```

- 問題点
  - これだけのために名前を2つ覚えなければならない
  - この目的以外に利用できない

関数型

- 関数型

```
public interface Function1Void<A> {
  void apply(A a);
}
```

- もし AWT ActionListener が関数型を使っていれば

```
class ButtonApp {
  private final Button button = new Button();

  public ButtonApp() {
    button.addActionListener(
     new Function1Void<ActionEvent>() {
      public void apply(ActionEvent e) {
        System.out.println("event received: "+e);
      }});}}
```

- ポイント
  - 覚える名前は2つだが、他の目的にも使える

無名関数

- さらにもし Java が無名関数を提供すれば

```
public ButtonApp() {
  private final Button button = new Button();

  button.addActionListener(#{ ActionEvent e ->
    System.out.println("event received: "+e)
  });
}
```

- ポイント
  - 構文を覚えるだけ
  - 覚えるべき名前はない


ムダなことを
覚えないといけないのは
抽象化に失敗しているから


無名関数は
イベントハンドラや高階関数
のための強力な抽象化

さようなら for

例題

- 入力として整数のリストあるいは配列
{ 10, 20, 30, 40, 50 } がある
- 0 から数えて n 番目の要素には n を掛ける
- それらをすべて足し合わせる

- つまり、以下のような計算をする

10 * 0 + 20 * 1 + 30 * 2 + 40 * 3 + 50 *4 = 400

10 * 0 + 20 * 1 + 30 * 2 + 40 * 3 + 50 *4 = 400

Java プログラマなら
for 文か類似のループで
問題を解く

不格好な for 文

- Douglas Crockford のエッセイ
  「JavaScript: 世界で最も誤解されているプログラミング言語」
  http://www.crockford.com/javascript/javascript.html

波括弧や不格好な for 文がある
JavaScript の C ライクな文法を見ると、
通常の命令型言語のように思える。

これは誤解を与えやすい。
なぜなら、JavaScript は、
C や Java とよりも
関数型言語 Lisp や Scheme との方が
共通点が多いからだ。


for 文って不格好なの？

for の秘密

- 山本和彦は、for について授業で熱く語っていた
- for の意味
  - for は期間の for だ！
  - 例) for two days
- 非対称範囲を使え！
  - `for (i = 0; i < N; i++) { }`
  - 左の境界は入るが、右の境界は入らない
  - 0 から始めると配列と相性がよい
  - N をそのまま使える
  - 個数 = 最後 - 最初 + 1
    - (N - 1) - 0 + 1 = N
  - 不等号を使うと安全性
    - コンピュータでは、1/3 + 1/3 + 1/3 ≠ 1

Java で不格好な for

```
public static int func(int[] ar) {
  int ret = 0;
  for (int i = 0; i < ar.length; i++) {
    ret = ret + ar[i] * i;
  }
  return ret;
}

int[] inp = {10,20,30,40,50};
func(inp);
→ 400
```

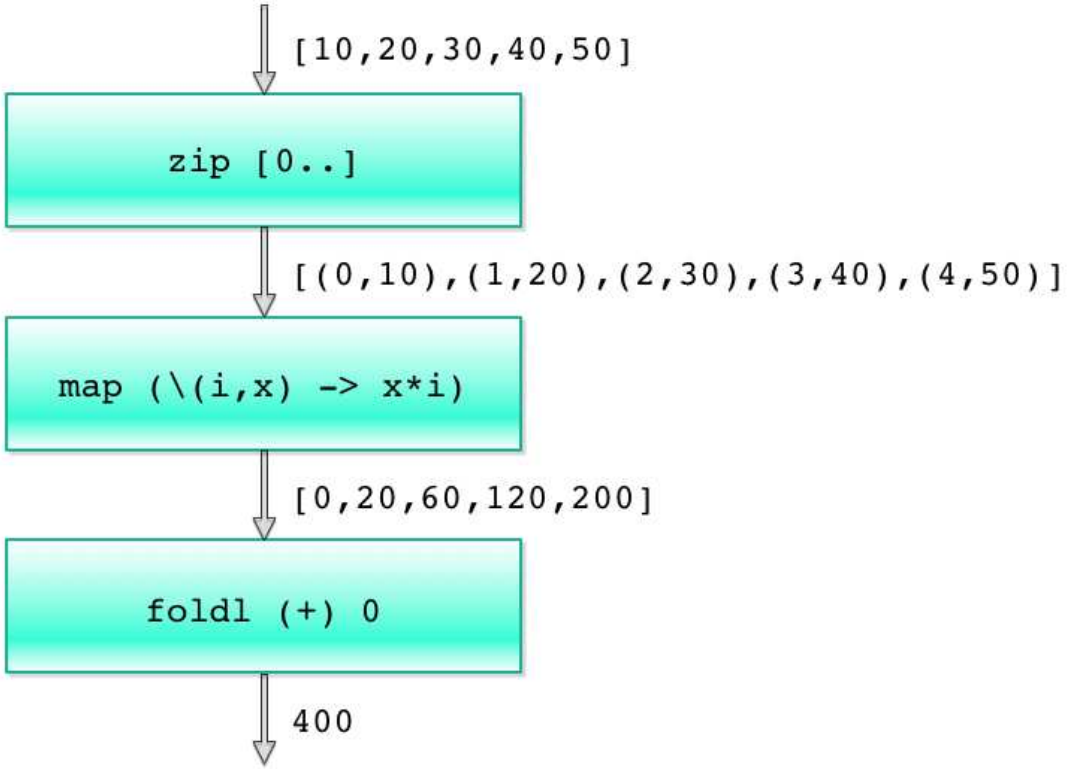
Haskell で MapReduce

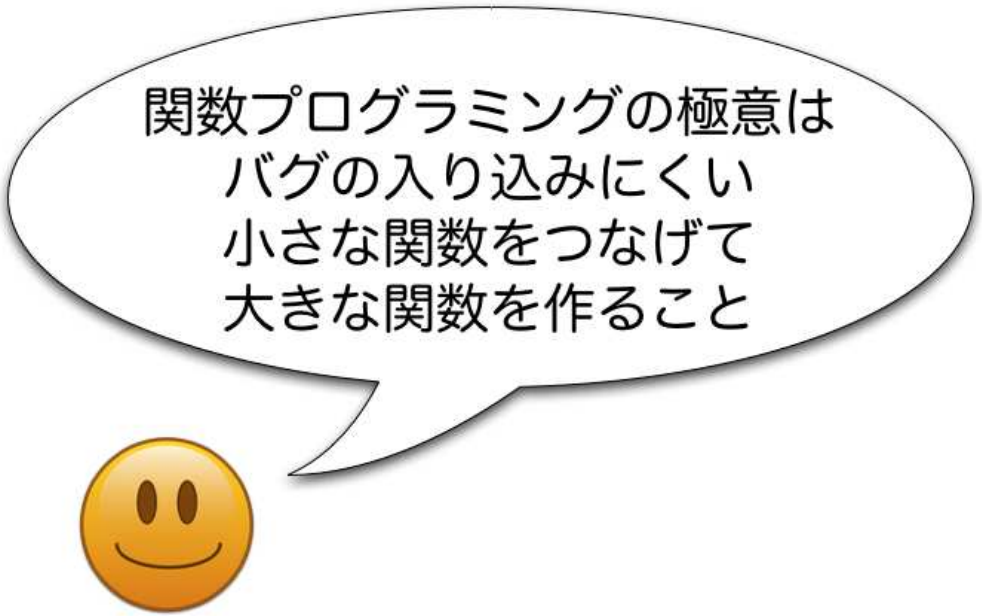
```
zip [0..] [10,20,30,40,50]
→ [(0,10),(1,20),(2,30),(3,40),(4,50)]

map (\(i,x) -> x*i) 上の式
→ [0,20,60,120,200]

foldl (+) 0 上の式
→ ((((0 + 0) + 20) + 60) + 120) + 200
→ 400
```

- 関数を合成する

```
func = foldl (+) 0
     . map (\(i,x) -> x*i)
     . zip [0..]

func [10,20,30,40,50]
→ 400
```

部品プログラミング


[10,20,30,40,50]
zip [0..]
[(0,10),(1,20),(2,30),(3,40),(4,50)]
map (\(i,x) -> x*i)
[0,20,60,120,200]
foldl (+) 0
400

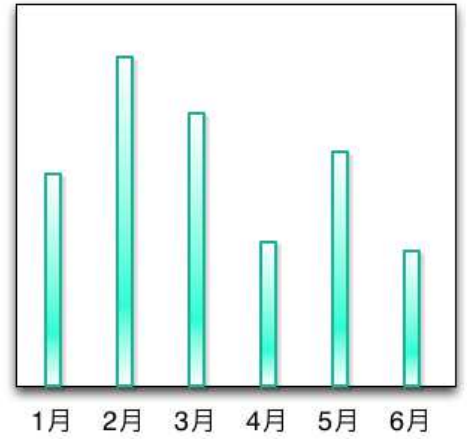

関数プログラミングの極意は
バグの入り込みにくい
小さな関数をつなげて
大きな関数を作ること

部品プログラミング

月ごとの売り上げの集計

- 「函数プログラミングのエッセンスと考え方」より
  - 小笠原啓
    - http://www.itpl.co.jp/tech/func/essense_of_fp(sea0305).pdf

2013年1月20日	2,020円
2013年1月23日	1,750円
2013年2月08日	650円
2013年2月14日	3,740円
2013年2月18日	8,000円
2013年2月27日	1,960円
2013年3月02日	2,200円
2013年3月16日	6,540円



Java で Sale を定義

```
public class Sale {
  Date date;
  int amount;

  Sale (Date _date, int _amount) {
    date = _date;
    amount = _amount;
  }

  // Date から月を取り出す
  public static int getMonth (Date d) {
    Calendar cal = Calendar.getInstance();
    cal.setTime(d);
    int month = cal.get(Calendar.MONTH);
    return month;
  }
}
```

Java で月ごとの売り上げ

```
public static List<Integer>
monthlySale(List<Sale> sales) {
  ArrayList<Integer> result =
    new ArrayList<Integer>();
  int sum = 0;
  int prevMonth = getMonth(sales.get(0).date);
  for (Sale sale : sales) {
    if(prevMonth == getMonth(sale.date)) {
      sum += sale.amount;
    } else {
      result.add(sum);
      sum = sale.amount;
    }
    prevMonth = getMonth(sale.date);
  }
  if(sum != 0) {
    result.add(sum);
  }
  return result;
}
```

Haskell で Sale の定義

```
data Sale = Sale UTCTime Int

time :: Sale -> UTCTime
time   (Sale t _) = t

amount :: Sale -> Int
amount (Sale _ m) = m

month :: Sale -> Int
month s = m
  where
    day = utctDay (time s)
    (_, m, _) = toGregorian day
```

月ごとにまとめる

```
groupBy ((==) `on` month)
  [Sale 2013-01-20 2020
  ,Sale 2013-01-23 1750
  ,Sale 2013-02-08  650
  ,Sale 2013-02-14 3740
  ,Sale 2013-02-18 8000
  ...
→
  [[Sale 2013-01-20 2020
   ,Sale 2013-01-23 1750
   ]
  ,[Sale 2013-02-08  650
   ,Sale 2013-02-14 3740
   ,Sale 2013-02-18 8000
  ...
```

- UTCTime のリテラルは実際とは異なります

売り上げを取り出す

```
map (map amount)
  [[Sale 2013-01-20 2020
   ,Sale 2013-01-23 1750
   ]
  ,[Sale 2013-02-08  650
   ,Sale 2013-02-14 3740
   ,Sale 2013-02-18 8000
  ...
→
  [[2020,1750]
  ,[650,3740,8000,1960]
  ,[2200,6540...
```

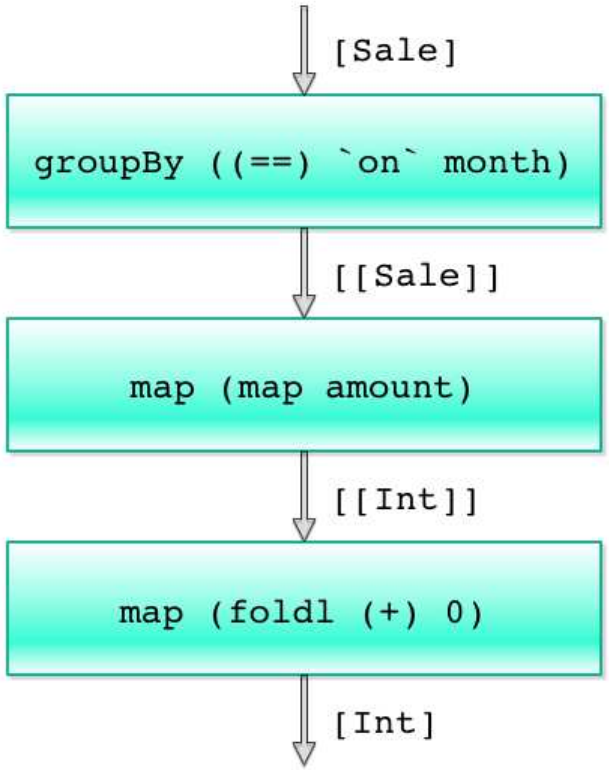
売り上げを足す

```
map (foldl (+) 0)
  [[2020,1750]
  ,[650,3740,8000,1960]
  ,[2200,6540...
```

→ `[3770,14350,...`

Haskell で月ごとの売り上げ

```
monthlySale :: [Sale] -> [Int]
monthlySale = map (foldl (+) 0)
            . map (map amount)
            . groupBy ((==) `on` month)
```




一枚岩プログラミングより
部品プログラミング

さようなら null

Java でお母さんDB

```
public class Person {
  String name;
  int age;
}


HashMap<String,Person> db
  = new HashMap<String,Person>();

Person sazae = new Person("サザエ", 24);
Person fune  = new Person("フネ", 52);

map.put("タラオ",sazae);
map.put("サザエ",fune);
//"フネ" のお母さんは亡くなっている
```

おばあちゃんを探せ

```
String me = "タラオ";
Person mother = db.get(me);           // "サザエ"
Person gramma = db.get(mother.name); // "フネ"

String me = "サザエ";
Person mother = db.get(me);           // "フネ"
Person gramma = db.get(mother.name); // null

String me = "フネ";
Person mother = db.get(me);           // null
Person gramma = db.get(mother.name); // 例外
```

安全におばあちゃんを探せ

```
String me = "フネ";
Person mother = db.get(me);
Person gramma = null;
if (mother != null) {
  gramma = db.get(mother.name);
}
```

null の問題点

```
<Person> get(<String>)
```

- null は、いろんな型になれる
- 型から null を返すか分からない
  - 返すかもしれないし
  - 返さないかもしれない
- Java プログラマは必ず null を踏む
  - コンパイラの支援がないので null の処理を忘れる

nullは
10億ドルの失敗

Tony Hoare

Haskell でお母さんDB

```
data Person = Person String Int

name :: Person -> String
name (Person n _) = n

age :: Person -> Int
age  (Person _ a) = a

db :: [(String, Person)]
db = [("タラオ", Person "サザエ" 24)
     ,("サザエ", Person "フネ" 52)]
```

lookup と Maybe

- DB の検索関数

```
lookup :: String
       -> [(String, Person)]
       -> Maybe Person
```

- Person は、失敗しない型
- Maybe Person は、失敗するかもしれない型
  - 答えなし: `Nothing`
  - 答えあり: `Just (Person "サザエ" 24)`

安全におばあちゃんを探せ

- Maybe Person はパターンマッチで処理
  - case ... of 式
  - すべての場合を網羅しないとコンパイラが警告を出す

```
findGramma :: String -> Maybe Person
findGramma me = case lookup me db of
    Nothing     -> Nothing
    Just mother -> lookup (name mother) db
```


関数の振る舞いは
適切な型で表現しよう

地下配線

安全にひいおばあちゃんを探せ

- おばあちゃんを探せ (Java 版再掲)

```
String me = "フネ";
Person mother = db.get(me);
Person gramma = null;
if (mother != null) {
  gramma = db.get(mother.name);
}
```

- ひいおばあちゃんを探せ

```
String me = "フネ";
Person mother = db.get(me);
Person gramma = null;
Person ggramma = null;
if (mother != null) {
   gramma = db.get(mother.name);
  if (gramma != null) {
    ggramma = db.get(gramma.name);
  }
}
```

安全にひいおばあちゃんを探せ

- おばあちゃんを探せ(Haskell 版再掲)

```
findGramma :: String -> Maybe Person
findGramma me = case lookup me db of
    Nothing     -> Nothing
    Just mother -> lookup (name mother) db
```

- ひいおばあちゃんを探せ

```
findGGramma :: String -> Maybe Person
findGGramma me = case lookup me db of
    Nothing     -> Nothing
    Just mother -> case lookup (name mother) db of
        Nothing     -> Nothing
        Just gramma -> lookup (name gramma) db
```

問題点

- 入れ子構造
- エラー処理にロジックが埋もれている

Maybe とは何なのか？

- Maybe は「直和型」の一種
  - Java の列挙型は直和型の一種
- 直和型
  - 「または」を表す型
- 真理値

```
data Bool = False | True
```

- 成功に値がある

```
data Maybe a = Nothing | Just a
```

- 成功にも失敗にも値がある

```
data Either l r = Left l | Right r
```

演算の直列化

- 真理値は直列にできる

```
isAlive mother && isAlive gramma
```

- Maybe a も直列にできないか？

```
findMother me &&> findMother mother
```

地下配線

- 地下配線 &&>

```
mx &&> f = case mx of
    Nothing -> Nothing
    Just x  -> f x
```

- 直列版

```
findGGramma :: String -> Maybe Person
findGGramma me =
  lookup me db
  &&> \mother -> lookup (name mother) db
  &&> \gramma -> lookup (name gramma) db
```

- 入れ子版(再掲)

```
findGGramma :: String -> Maybe Person
findGGramma me = case lookup me db of
    Nothing     -> Nothing
    Just mother -> case lookup (name mother) db of
        Nothing     -> Nothing
        Just gramma -> lookup (name gramma) db
```


エラー処理は
地下配線に押し込めよう

豊かなデータ型

直積型

- 直積型
  - 「かつ」を表す型
  - Java のクラス

```
data Person = Person String Int
data Sale = Sale UTCTime Int
```

代数データ型

- 直積の直和

```
data Shape = Rectangle Side Side
           | Ellipse Radius Radius
```

- 再帰も含む直積の直和

```
data IntList = Nil
             | Cons Int IntList

data Tree a = Leaf
            | Node (Tree a) a (Tree a)
```

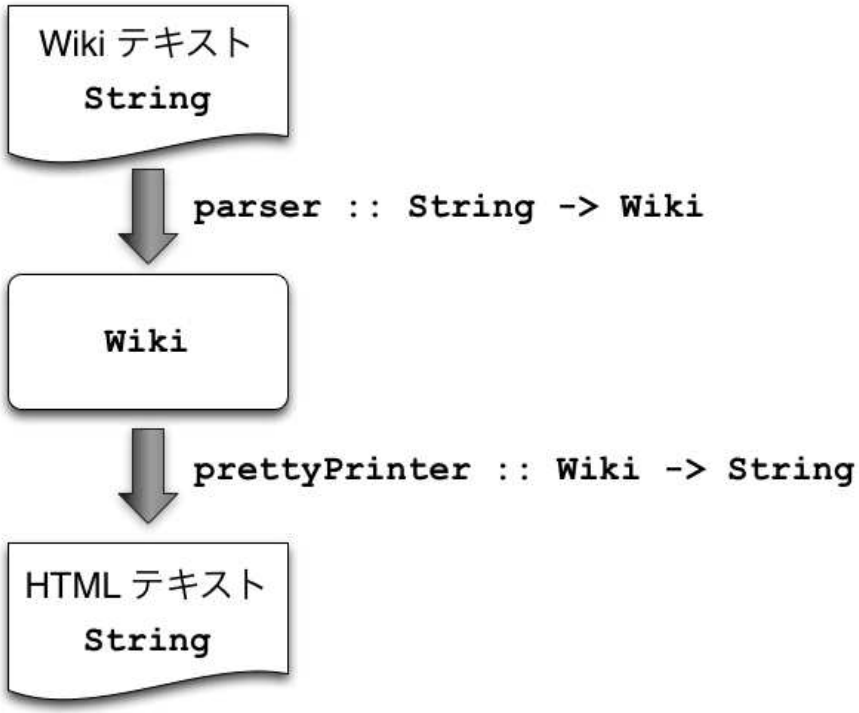
例) Wiki を作る

```
* 大見出し

行頭にキー文字がなければ段落

============== 罫線
** 中見出し

\* で始まる段落

リンクは [タイトル URL] と書く

-箇条書き レベル1
--箇条書き レベル2
--箇条書き レベル2
-箇条書き レベル1
++番号付き箇条書き レベル2
++番号付き箇条書き レベル2
```

Wiki のデータ構造

```
type Wiki = [Element]

data Element = HR
             | H Int HText
             | P HText
             | UOL Xlist

type HText = [PText]

data PText = Raw Char
           | Escaped Char  -- backslash
           | Anchor Title URL

type URL = String

type Title = String

data Xlist = Ulist [Xitem]
           | Olist [Xitem]
           | Empty

data Xitem = Item HText Xlist
```

データ型を中心にプログラミングする




豊かなデータ型に対して
プログラミングしよう